Linked List Basics

By Nick Parlante

Abstract

This document introduces the basic structures and techniques for building linked lists with a mixture of explanations, drawings, sample code, and exercises. The material is useful if you want to understand linked lists or if you want to see a realistic, applied example of pointer-intensive code. A separate document, Linked List Problems (http://cslibrary.stanford.edu/105/), presents 18 practice problems covering a wide range of difficulty.

Linked lists are useful to study for two reasons. Most obviously, linked lists are a data structure which you may want to use in real programs. Seeing the strengths and weaknesses of linked lists will give you an appreciation of the some of the time, space, and code issues which are useful to thinking about any data structures in general.

Somewhat less obviously, linked lists are great way to learn about pointers. In fact, you may never use a linked list in a real program, but you are certain to use lots of pointers. Linked list problems are a nice combination of algorithms and pointer manipulation. Traditionally, linked lists have been the domain where beginning programmers get the practice to really understand pointers.

Audience

The article assumes a basic understanding of programming and pointers. The article uses C syntax for its examples where necessary, but the explanations avoid C specifics as much as possible — really the discussion is oriented towards the important concepts of pointer manipulation and linked list algorithms.

Other Resources

- Link List Problems (http://cslibrary.stanford.edu/105/) Lots of linked list problems, with explanations, answers, and drawings. The "problems" article is a companion to this "explanation" article.
- Pointers and Memory (http://cslibrary.stanford.edu/102/) Explains all about how pointers and memory work. You need some understanding of pointers and memory before you can understand linked lists.
- Essential C (http://cslibrary.stanford.edu/101/) Explains all the basic features of the C programming language.

Contents

Edition

Originally 1998 there was just one "Linked List" document that included a basic explanation and practice problems. In 1999, it got split into two documents: #103 (this document) focuses on the basic introduction, while #105 is mainly practice problems. This 4-12-2001 edition represents minor edits on the 1999 edition.

Dedication

This document is distributed for free for the benefit and education of all. That a person seeking knowledge should have the opportunity to find it. Thanks to Stanford and my boss Eric Roberts for supporing me in this project. Best regards, Nick -- nick.parlante@cs.stanford.edu

Section 1 — Linked List Basics

Why Linked Lists?

Linked lists and arrays are similar since they both store collections of data. The terminology is that arrays and linked lists store "elements" on behalf of "client" code. The specific type of element is not important since essentially the same structure works to store elements of any type. One way to think about linked lists is to look at how arrays work and think about alternate approaches.

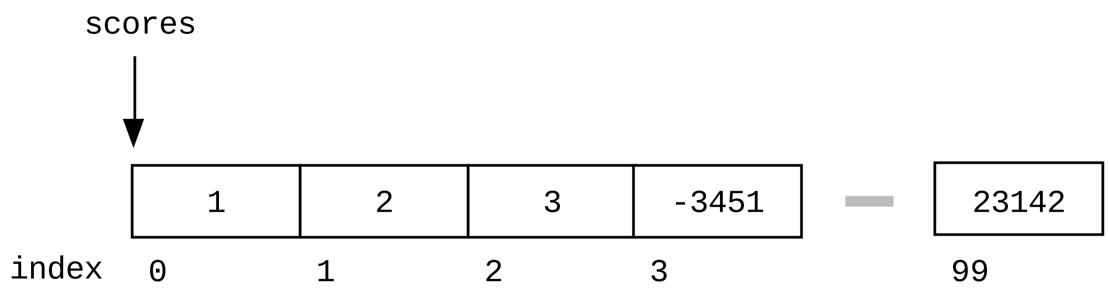
Array Review

Arrays are probably the most common data structure used to store collections of elements. In most languages, arrays are convenient to declare and the provide the handy [ ] syntax to access any element by its index number. The following example shows some typical array code and a drawing of how the array might look in memory. The code allocates an array `int scores[100]`, sets the first three elements set to contain the numbers 1, 2, 3 and leaves the rest of the array uninitialized...

```
void ArrayTest() {
   int scores[100];

   // operate on the elements of the scores array...
   scores[0] = 1;
   scores[1] = 2;
   scores[2] = 3;
}
```

Here is a drawing of how the scores array might look like in memory. The key point is that the entire array is allocated as one block of memory. Each element in the array gets its own space in the array. Any element can be accessed directly using the [ ] syntax.



Once the array is set up, access to any element is convenient and fast with the [ ] operator. (Extra for experts) Array access with expressions such as `scores[i]` is almost always implemented using fast address arithmetic: the address of an element is computed as an offset from the start of the array which only requires one multiplication and one addition.

The disadvantages of arrays are...

1) The size of the array is fixed — 100 elements in this case. Most often this size is specified at compile time with a simple declaration such as in the example above . With a little extra effort, the size of the array can be deferred until the array is created at runtime, but after that it remains fixed. (extra for experts) You can go to the trouble of dynamically allocating an array in the heap and then dynamically resizing it with realloc(), but that requires some real programmer effort.

2) Because of (1), the most convenient thing for programmers to do is to allocate arrays which seem "large enough" (e.g. the 100 in the scores example). Although convenient, this strategy has two disadvantages: (a) most of the time there are just 20 or 30 elements in the array and 70% of the space in the array really is wasted. (b) If the program ever needs to process more than 100 scores, the code breaks. A surprising amount of commercial code has this sort of naive array allocation which wastes space most of the time and crashes for special occasions. (Extra for experts) For relatively large arrays (larger than 8k bytes), the virtual memory system may partially compensate for this problem, since the "wasted" elements are never touched.

3) (minor) Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.

Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. The array's features all follow from its strategy of allocating the memory for all its elements in one block of memory. Linked lists use an entirely different strategy. As we will see, linked lists allocate memory for each element separately and only when necessary.

Pointer Refresher

Here is a quick review of the terminology and rules for pointers. The linked list code to follow will depend on these rules. (For much more detailed coverage of pointers and memory, see Pointers and Memory, http://cslibrary.stanford.edu/102/).

- *Pointer/Pointee* A "pointer" stores a reference to another variable sometimes known as its "pointee". Alternately, a pointer may be set to the value NULL which encodes that it does not currently refer to a pointee. (In C and C++ the value NULL can be used as a boolean false).

- *Dereference* The dereference operation on a pointer accesses its pointee. A pointer may only be dereferenced after it has been set to refer to a specific pointee. A pointer which does not have a pointee is "bad" (below) and should not be dereferenced.

- *Bad Pointer* A pointer which does not have an assigned a pointee is "bad" and should not be dereferenced. In C and C++, a dereference on a bad sometimes crashes immediately at the dereference and sometimes randomly corrupts the memory of the running program, causing a crash or incorrect computation later. That sort of random bug is difficult to track down. **In C and C++, all pointers start out with bad values**, so it is easy to use bad pointer accidentally. Correct code sets each pointer to have a good value before using it. Accidentally using a pointer when it is bad is the most common bug in pointer code. In Java and other runtime oriented languages, pointers automatically start out with the NULL value, so dereferencing one is detected immediately. Java programs are much easier to debug for this reason.

- *Pointer assignment* An assignment operation between two pointers like `p=q;` makes the two pointers point to the same pointee. It does not copy the pointee memory. After the assignment both pointers will point to the same pointee memory which is known as a "sharing" situation.

- *malloc()* malloc() is a system function which allocates a block of memory in the "heap" and returns a pointer to the new block. The prototype for malloc() and other heap functions are in `stdlib.h`. The argument to malloc() is the integer size of the block in bytes. Unlike local ("stack") variables, heap memory is not automatically deallocated when the creating function exits. malloc() returns NULL if it cannot fulfill the request. (extra for experts) You may check for the NULL case with assert() if you wish just to be safe. Most modern programming systems will throw an exception or do some other automatic error handling in their memory allocator, so it is becoming less common that source code needs to explicitly check for allocation failures.

- *free()* free() is the opposite of malloc(). Call free() on a block of heap memory to indicate to the system that you are done with it. The argument to free() is a pointer to a block of memory in the heap — a pointer which some time earlier was obtained via a call to malloc().

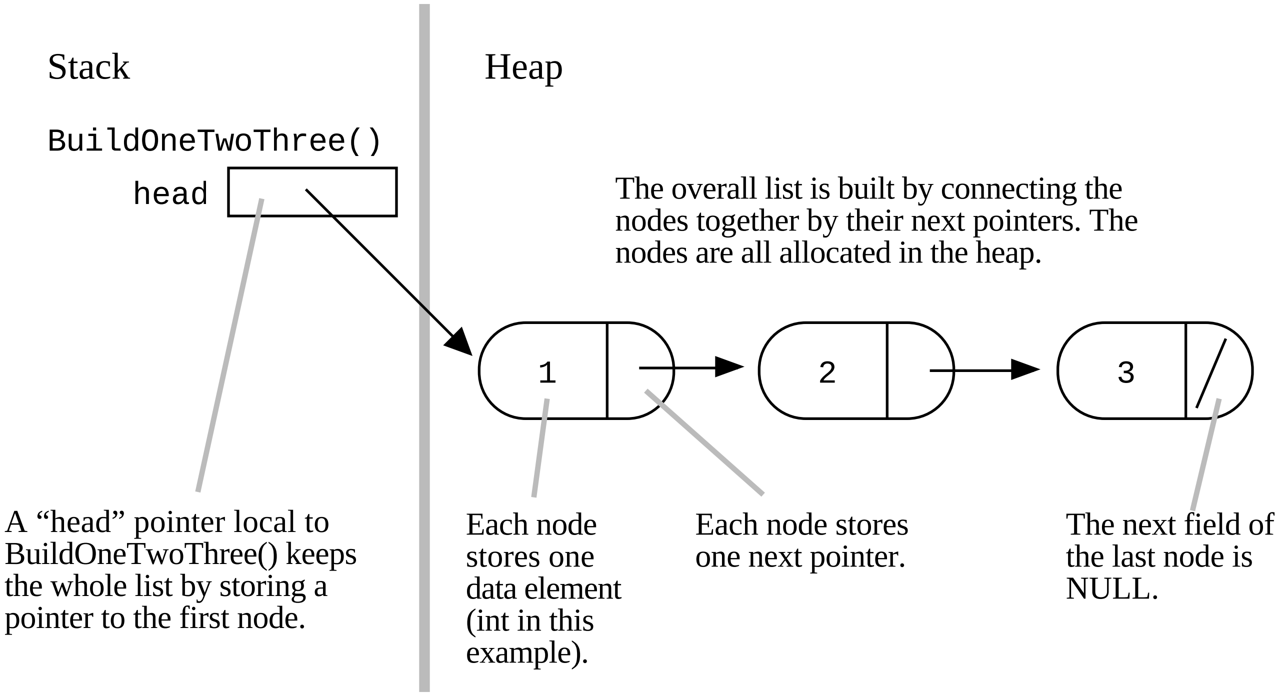
What Linked Lists Look Like

An array allocates memory for all its elements lumped together as one block of memory. In contrast, a linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node". The list gets is overall structure by using pointers to connect all its nodes together like the links in a chain.

Each node contains two fields: a "data" field to store whatever element type the list holds for its client, and a "next" field which is a pointer used to link one node to the next node. Each node is allocated in the heap with a call to malloc(), so the node memory continues to exist until it is explicitly deallocated with a call to free(). The front of the list is a

pointer to the first node. Here is what a list containing the numbers 1, 2, and 3 might look like...

The Drawing Of List {1, 2, 3}



This drawing shows the list built in memory by the function BuildOneTwoThree() (the full source code for this function is below). The beginning of the linked list is stored in a "head" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its .next field set to NULL to mark the end of the list. Code can access any node in the list by starting at the head and following the `.next` pointers. Operations towards the front of the list are fast while operations which access node farther down the list take longer the further they are from the front. This "linear" cost to access a node is fundamentally more costly then the constant time [ ] access provided by arrays. In this respect, linked lists are definitely less efficient than arrays.

Drawings such as above are important for thinking about pointer code, so most of the examples in this article will associate code with its memory drawing to emphasize the habit. In this case the head pointer is an ordinary local pointer variable, so it is drawn separately on the left to show that it is in the stack. The list nodes are drawn on the right to show that they are allocated in the heap.

The Empty List — NULL

The above is a list pointed to by head is described as being of "length three" since it is made of three nodes with the .next field of the last node set to NULL. There needs to be some representation of the empty list — the list with zero nodes. The most common representation chosen for the empty list is a NULL head pointer. The empty list case is the one common weird "boundary case" for linked list code. All of the code presented in this article works correctly for the empty list case, but that was not without some effort. When working on linked list code, it's a good habit to remember to check the empty list case to verify that it works too. Sometimes the empty list case works the same as all the cases, but sometimes it requires some special case code. No matter what, it's a good case to at least think about.

Linked List Types: Node and Pointer

Before writing the code to build the above list, we need two data types...

- *Node* The type for the nodes which will make up the body of the list. These are allocated in the heap. Each node contains a single client data element and a pointer to the next node in the list. Type: `struct node`

  ```
  struct node {
     int            data;
     struct node*   next;
  };
  ```

- *Node Pointer* The type for pointers to nodes. This will be the type of the head pointer and the .next fields inside each node. In C and C++, no separate type declaration is required since the pointer type is just the node type followed by a '*'. Type: `struct node*`

BuildOneTwoThree() Function

Here is simple function which uses pointer operations to build the list {1, 2, 3}. The memory drawing above corresponds to the state of memory at the end of this function. This function demonstrates how calls to malloc() and pointer assignments (=) work to build a pointer structure in the heap.

```
/*
 Build the list {1, 2, 3} in the heap and store
 its head pointer in a local stack variable.
 Returns the head pointer to the caller.
*/
struct node* BuildOneTwoThree() {
   struct node* head = NULL;
   struct node* second = NULL;
   struct node* third = NULL;

   head = malloc(sizeof(struct node));    // allocate 3 nodes in the heap
   second = malloc(sizeof(struct node));
   third = malloc(sizeof(struct node));

   head->data = 1;                  // setup first node
   head->next = second;       // note: pointer assignment rule

   second->data = 2;                // setup second node
   second->next = third;

   third->data = 3;                 // setup third link
   third->next = NULL;

   // At this point, the linked list referenced by "head"
   // matches the list in the drawing.
   return head;
}
```

Exercise

Q: Write the code with the smallest number of assignments (=) which will build the above memory structure. A: It requires 3 calls to malloc(). 3 int assignments (=) to setup the ints. 4 pointer assignments to setup head and the 3 next fields. With a little cleverness and knowledge of the C language, this can all be done with 7 assignment operations (=).

Length() Function

The Length() function takes a linked list and computes the number of elements in the list. Length() is a simple list function, but it demonstrates several concepts which will be used in later, more complex list functions...

```
/*
 Given a linked list head pointer, compute
 and return the number of nodes in the list.
*/
int Length(struct node* head) {
   struct node* current = head;
   int count = 0;

   while (current != NULL) {
      count++;
      current = current->next;
   }

   return count;
}
```

There are two common features of linked lists demonstrated in Length()...

1) Pass The List By Passing The Head Pointer

The linked list is passed in to Length() via a single head pointer. The pointer is copied from the caller into the "head" variable local to Length(). Copying this pointer does not duplicate the whole list. It only copies the pointer so that the caller and Length() both have pointers to the same list structure. This is the classic "sharing" feature of pointer code. Both the caller and length have copies of the head pointer, but they share the pointee node structure.

2) Iterate Over The List With A Local Pointer

The code to iterate over all the elements is a very common idiom in linked list code....

```
struct node* current = head;
while (current != NULL) {
   // do something with *current node

   current = current->next;
}
```

The hallmarks of this code are...

1) The local pointer, `current` in this case, starts by pointing to the same node as the head pointer with `current = head;`. When the function exits, `current` is automatically deallocated since it is just an ordinary local, but the nodes in the heap remain.

2) The while loop tests for the end of the list with `(current != NULL)`. This test smoothly catches the empty list case — `current` will be NULL on the first iteration and the while loop will just exit before the first iteration.

3) At the bottom of the while loop, `current = current->next;` advances the local pointer to the next node in the list. When there are no more links, this sets the pointer to NULL. If you have some linked list

code which goes into an infinite loop, often the problem is that step (3) has been forgotten.

Calling Length()

Here's some typical code which calls Length(). It first calls BuildOneTwoThree() to make a list and store the head pointer in a local variable. It then calls Length() on the list and catches the int result in a local variable.

```
void LengthTest() {
   struct node* myList = BuildOneTwoThree();

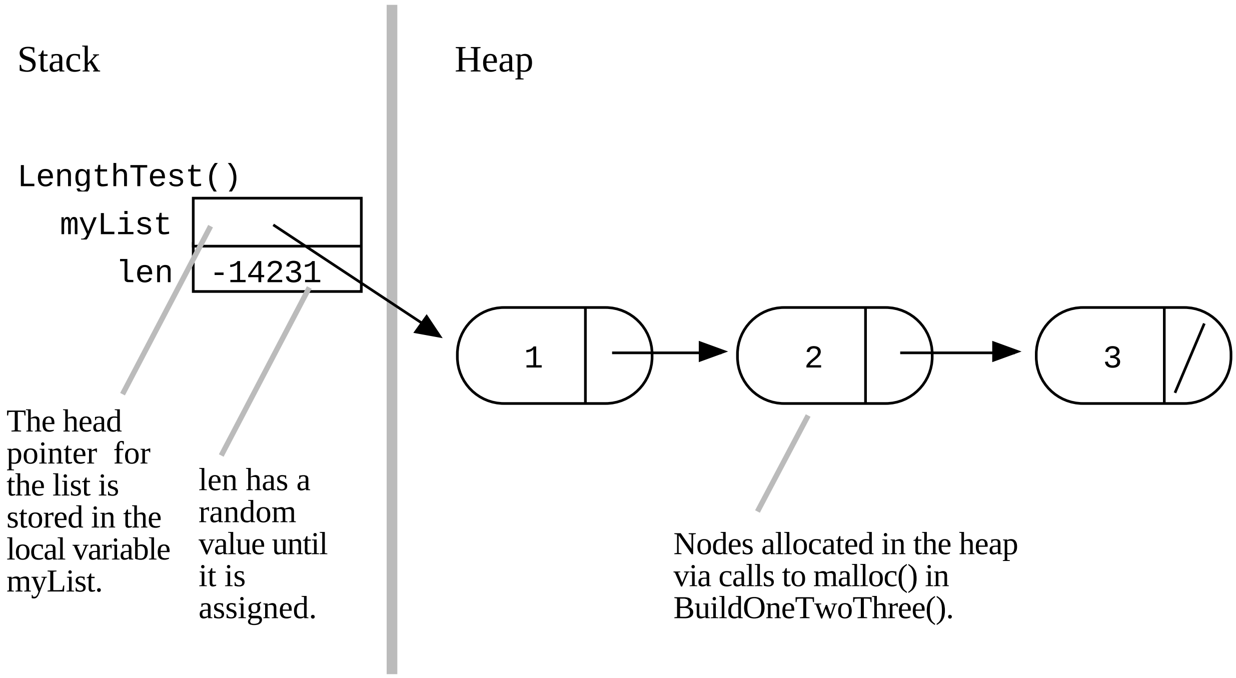
   int len = Length(myList);  // results in len == 3
}
```

Memory Drawings

The best way to design and think about linked list code is to use a drawing to see how the pointer operations are setting up memory. There are drawings below of the state of memory before and during the call to Length() — take this opportunity to practice looking at memory drawings and using them to think about pointer intensive code. You will be able to understand many of the later, more complex functions only by making memory drawings like this on your own.

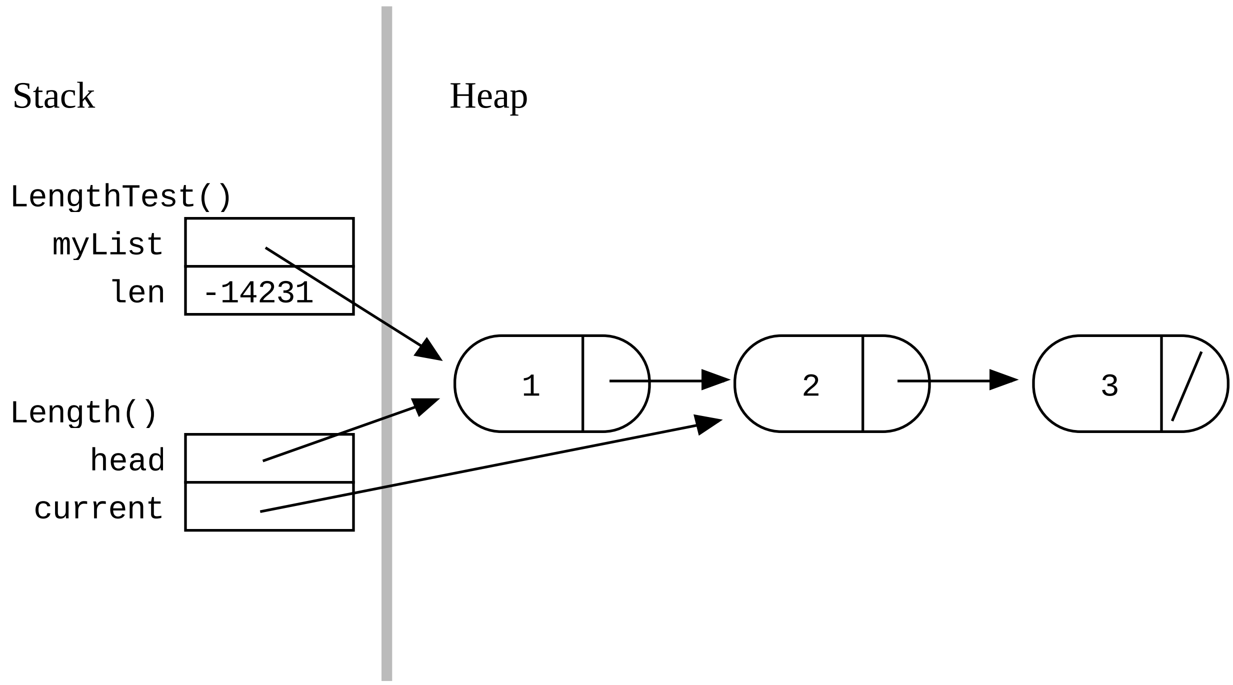
Start with the Length() and LengthTest() code and a blank sheet of paper. Trace through the execution of the code and update your drawing to show the state of memory at each step. Memory drawings should distinguish heap memory from local stack memory. Reminder: malloc() allocates memory in the heap which is only be deallocated by deliberate calls to free(). In contrast, local stack variables for each function are automatically allocated when the function starts and deallocated when it exits. Our memory drawings show the caller local stack variables above the callee, but any convention is fine so long as you realize that the caller and callee are separate. (See cslibrary.stanford.edu/102/, Pointers and Memory, for an explanation of how local memory works.)

Drawing 1 : Before Length()
Below is the state of memory just before the call to Length() in LengthTest() above. BuildOneTwoThree() has built the {1, 2, 3} list in the heap and returned the head pointer. The head pointer has been caught by the caller and stored in its local variable `myList`. The local variable len has a random value — it will only be given the value 3 when then call to Length() returns.



Drawing 2: Mid Length

Here is the state of memory midway through the execution of Length(). Length()'s local variables `head` and `current` have been automatically allocated. The `current` pointer started out pointing to the first node, and then the first iteration of the while loop advanced it to point to the second node.



Notice how the local variables in Length() (`head` and `current`) are separate from the local variables in LengthTest() (`myList` and `len`). The local variables `head` and `current` will be deallocated (deleted) automatically when Length() exits. This is fine — the heap allocated links will remain even though stack allocated pointers which were pointing to them have been deleted.

Exercise

Q: What if we said `head = NULL;` at the end of Length() — would that mess up the `myList` variable in the caller? A: No. head is a local which was initialized with a copy of the actual parameter, but changes do not automatically trace back to the actual parameter. Changes to the local variables in one function do not affect the locals of another function.

Exercise

Q: What if the passed in list contains no elements, does Length() handle that case properly? A: Yes. The representation of the empty list is a NULL head pointer. Trace Length() on that case to see how it handles it.

Section 2 — List Building

BuildOneTwoThree() is a fine as example of pointer manipulation code, but it's not a general mechanism to build lists. The best solution will be an independent function which adds a single new node to any list. We can then call that function as many times as we want to build up any list. Before getting into the specific code, we can identify the classic 3-Step Link In operation which adds a single node to the front of a linked list. The 3 steps are...

1) *Allocate* Allocate the new node in the heap and set its .data to whatever needs to be stored.
```
struct node* newNode;
newNode = malloc(sizeof(struct node));
newNode->data = data_client_wants_stored;
```

2) *Link Next* Set the .next pointer of the new node to point to the current first node of the list. This is actually just a pointer assignment — remember: "assigning one pointer to another makes them point to the same thing."
```
newNode->next = head;
```

3) *Link Head* Change the head pointer to point to the new node, so it is now the first node in the list.
```
head = newNode;
```

3-Step Link In Code

The simple LinkTest() function demonstrates the 3-Step Link In...

```
void LinkTest() {
   struct node* head = BuildTwoThree(); // suppose this builds the {2, 3} list
   struct node* newNode;

   newNode= malloc(sizeof(struct node));  // allocate
   newNode->data = 1;

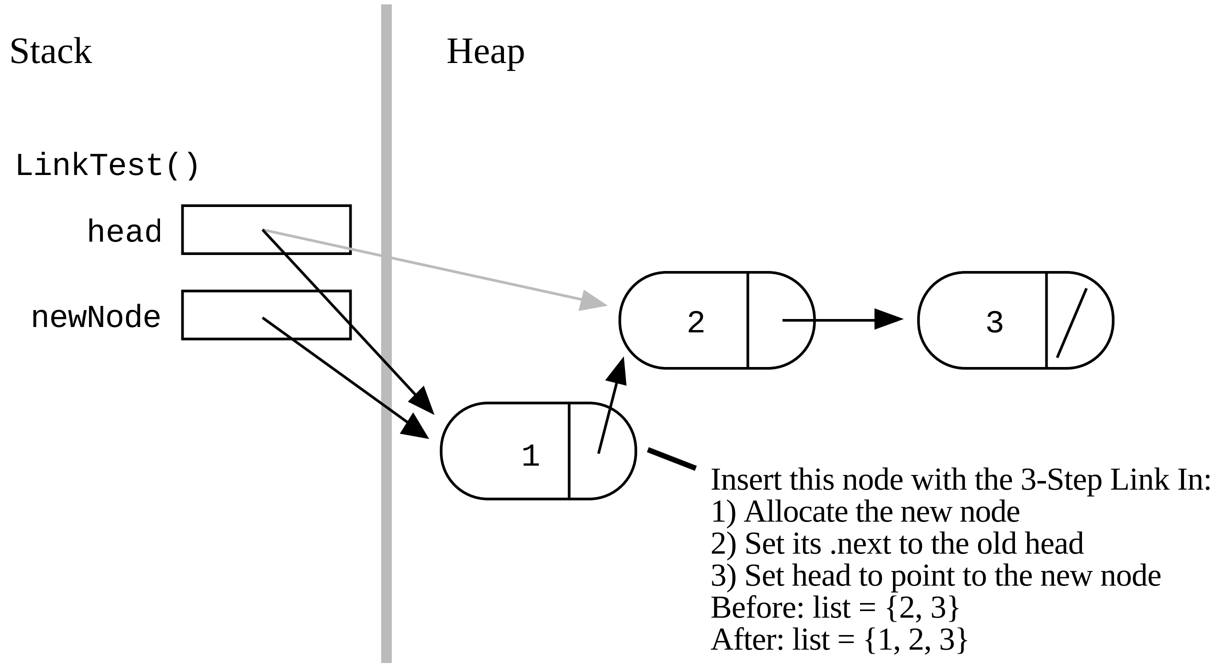
   newNode->next = head;                        // link next

   head = newNode;                              // link head

   // now head points to the list {1, 2, 3}
}
```

3-Step Link In Drawing

The drawing of the above 3-Step Link like (overwritten pointer values are in gray)...



Push() Function

With the 3-Step Link In in mind, the problem is to write a general function which adds a single node to head end of any list. Historically, this function is called "Push()" since we're adding the link to the head end which makes the list look a bit like a stack. Alternately it could be called InsertAtFront(), but we'll use the name Push().

WrongPush()

Unfortunately Push() written in C suffers from a basic problem: what should be the parameters to Push()? This is, unfortunately, a sticky area in C. There's a nice, obvious way to write Push() which looks right but is wrong. Seeing exactly how it doesn't work will provide an excuse for more practice with memory drawings, motivate the correct solution, and just generally make you a better programmer....

```
void WrongPush(struct node* head, int data) {
   struct node* newNode = malloc(sizeof(struct node));

   newNode->data = data;
   newNode->next = head;
   head = newNode;       // NO this line does not work!
}


void WrongPushTest() {
   List head = BuildTwoThree();

   WrongPush(head, 1);  // try to push a 1 on front -- doesn't work
}
```

WrongPush() is very close to being correct. It takes the correct 3-Step Link In and puts it an almost correct context. The problem is all in the very last line where the 3-Step Link In dictates that we change the head pointer to refer to the new node. What does the line `head = newNode;` do in WrongPush()? It sets a head pointer, but not the right one. It sets the variable named head local to WrongPush(). It does not in any way change the variable named head we really cared about which is back in the caller WrontPushTest().

Exercise

Make the memory drawing tracing WrongPushTest() to see how it does not work. The key is that the line `head = newElem;` changes the `head` local to WrongPush() not the `head` back in WrongPushTest(). Remember that the local variables for WrongPush() and WrongPushTest() are separate (just like the locals for LengthTest() and Length() in the Length() example above).

Reference Parameters In C

We are bumping into a basic "feature" of the C language that changes to local parameters are never reflected back in the caller's memory. This is a traditional tricky area of C programming. We will present the traditional "reference parameter" solution to this problem, but you may want to consult another C resource for further information. (See Pointers and Memory (http://cslibrary.stanford.edu/102/) for a detailed explanation of reference parameters in C and C++.)

We need Push() to be able to change some of the caller's memory — namely the head variable. The traditional method to allow a function to change its caller's memory is to pass a pointer to the caller's memory instead of a copy. So in C, to change an `int` in the caller, pass a `int*` instead. To change a `struct fraction`, pass a `struct fraction*` intead. To change an `X`, pass an `X*`. So in this case, the value we want to change is `struct node*`, so we pass a `struct node**` instead. The two stars (**) are a little scary, but really it's just a straight application of the rule. It just happens that the value we want to change already has one star (*), so the parameter to change it has two (**). Or put another way: the type of the head pointer is "pointer to a struct node." In order to change that pointer, we need to pass a pointer to it, which will be a "pointer to a pointer to a struct node".

Instead of defining `WrongPush(struct node* head, int data);` we define `Push(struct node** headRef, int data);`. The first form passes a copy of the head pointer. The second, correct form passes a pointer to the head pointer. The rule is: to modify caller memory, pass a pointer to that memory. The parameter has the word "ref" in it as a reminder that this is a "reference" (`struct node**`) pointer to the head pointer instead of an ordinary (`struct node*`) copy of the head pointer.

Correct Push() Code

Here are Push() and PushTest() written correctly. The list is passed via a pointer to the head pointer. In the code, this amounts to use of '&' on the parameter in the caller and use of '*' on the parameter in the callee. Inside Push(), the pointer to the head pointer is named "headRef" instead of just "head" as a reminder that it is not just a simple head pointer..

```
/*
 Takes a list and a data value.
 Creates a new link with the given data and pushes
 it onto the front of the list.
 The list is not passed in by its head pointer.
 Instead the list is passed in as a "reference" pointer
 to the head pointer -- this allows us
 to modify the caller's memory.
*/
void Push(struct node** headRef, int data) {
   struct node* newNode = malloc(sizeof(struct node));

   newNode->data = data;
   newNode->next = *headRef;  // The '*' to dereferences back to the real head
   *headRef = newNode;        // ditto
}

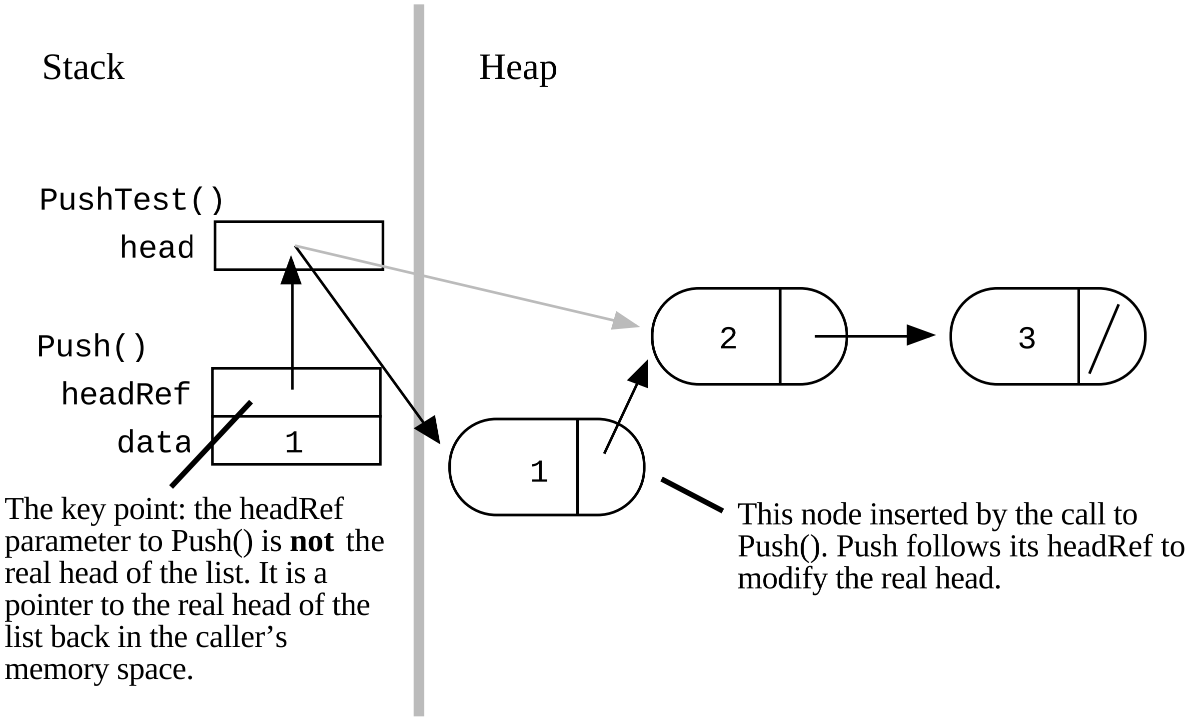

void PushTest() {
   struct node* head = BuildTwoThree();// suppose this returns the list {2, 3}

   Push(&head, 1);      // note the &
   Push(&head, 13);

   // head is now the list {13, 1, 2, 3}
}
```

Correct Push() Drawing

Here is a drawing of memory just before the first call to Push() exits. The original value of the head pointer is in gray. Notice how the `headRef` parameter inside Push() points back to the real `head` pointer back in PushTest(). Push() uses `*headRef` to access and change the real head pointer.



Exercise

The above drawing shows the state of memory at the end of the first call to Push() in PushTest(). Extend the drawing to trace through the second call to Push(). The result should be that the list is left with elements {13, 1, 2, 3}.

Exercise

The following function correctly builds a three element list using nothing but Push(). Make the memory drawing to trace its execution and show the final state of its list. This will also demonstrate that Push() works correctly for the empty list case.

```
void PushTest2() {
   struct node* head = NULL;  // make a list with no elements

   Push(&head, 1);
   Push(&head, 2);
   Push(&head, 3);

   // head now points to the list {3, 2, 1}
}
```

What About C++?

(Just in case you were curious) C++ has its built in "& argument" feature to implement reference parameters for the programmer. The short story is, append an '&' to the type of a parameter, and the compiler will automatically make the parameter operate by reference for you. The type of the argument is not disturbed by this — the types continue to act as

they appear in the source, which is the most convenient for the programmer. So In C++, Push() and PushTest() look like...

```
/*
 Push() in C++ -- we just add a '&' to the right hand
 side of the head parameter type, and the compiler makes
 that parameter work by reference. So this code changes
 the caller's memory, but no extra uses of '*' are necessary --
 we just access "head" directly, and the compiler makes that
 change reference back to the caller.
*/
void Push(struct node*& head, int data) {
   struct node* newNode = malloc(sizeof(struct node));

   newNode->data = data;
   newNode->next = head;      // No extra use of * necessary on head -- the compiler
   head = newNode;            // just takes care of it behind the scenes.
}


void PushTest() {
   struct node* head = BuildTwoThree();// suppose this returns the list {2, 3}

   Push(head, 1);       // No extra use & necessary -- the compiler takes
   Push(head, 13);      // care of it here too. Head is being changed by
                        // these calls.

   // head is now the list {13, 1, 2, 3}
}
```

The memory drawing for the C++ case looks the same as for the C case. The difference is that the C case, the *'s need to be taken care of in the code. In the C++ case, it's handled invisibly in the code.

Section 3 — Code Techniques

This section summarizes, in list form, the main techniques for linked list code. These techniques are all demonstrated in the examples in the next section.

1) Iterate Down a List

A very frequent technique in linked list code is to iterate a pointer over all the nodes in a list. Traditionally, this is written as a `while` loop. The head pointer is copied into a local variable `current` which then iterates down the list. Test for the end of the list with `current!=NULL`. Advance the pointer with `current=current->next`.

```
// Return the number of nodes in a list (while-loop version)
int Length(struct node* head) {
   int count = 0;
   struct node* current = head;

   while (current != NULL) {
      count++;
      current = current->next
   }

   return(count);
}
```

Alternately, some people prefer to write the loop as a `for` which makes the initialization, test, and pointer advance more centralized, and so harder to omit...

```
for (current = head; current != NULL; current = current->next) {
```

2) Changing a Pointer With A Reference Pointer

Many list functions need to change the caller's head pointer. To do this in the C language, pass a pointer to the head pointer. Such a pointer to a pointer is sometimes called a "reference pointer". The main steps for this technique are...

- Design the function to take a pointer to the head pointer. This is the standard technique in C — pass a pointer to the "value of interest" that needs to be changed. To change a `struct node*`, pass a `struct node**`.

- Use '&' in the caller to compute and pass a pointer to the value of interest.

- Use '*' on the parameter in the callee function to access and change the value of interest.

The following simple function sets a head pointer to NULL by using a reference parameter....

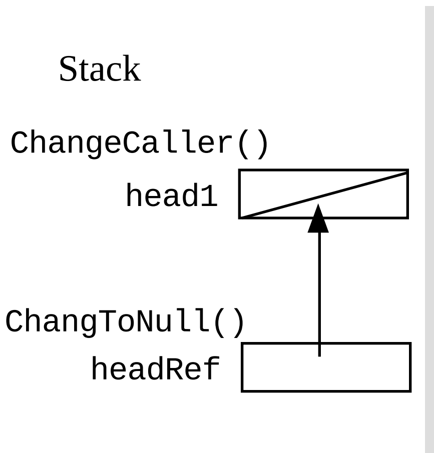
```
// Change the passed in head pointer to be NULL
// Uses a reference pointer to access the caller's memory
void ChangeToNull(struct node** headRef) {      // Takes a pointer to
                                                // the value of interest
```

```
   *headRef = NULL;     // use '*' to access the value of interest
}


void ChangeCaller() {
   struct node* head1;
   struct node* head2;

   ChangeToNull(&head1);      // use '&' to compute and pass a pointer to
   ChangeToNull(&head2);      // the value of interest
   // head1 and head2 are NULL at this point
}
```

Here is a drawing showing how the `headRef` pointer in ChangeToNull() points back to the variable in the caller...



See the use of Push() above and its implementation for another example of reference pointers.

3) Build — At Head With Push()

The easiest way to build up a list is by adding nodes at its "head end" with Push(). The code is short and it runs fast — lists naturally support operations at their head end. The disadvantage is that the elements will appear in the list in the reverse order that they are added. If you don't care about order, then the head end is the best.

```
struct node* AddAtHead() {
   struct node* head = NULL;
   int i;

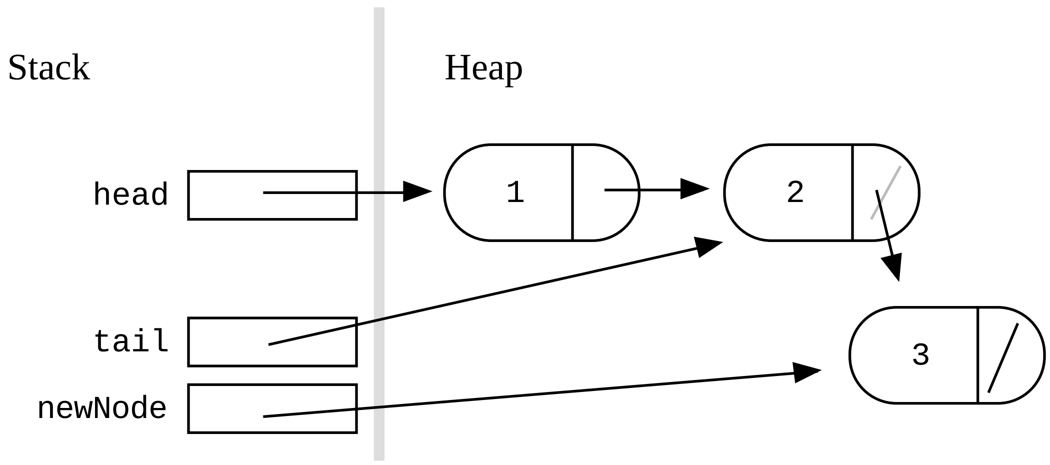
   for (i=1; i<6; i++) {
      Push(&head, i);
   }

   // head == {5, 4, 3, 2, 1};
   return(head);
}
```

4) Build — With Tail Pointer

What about adding nodes at the "tail end" of the list? Adding a node at the tail of a list most often involves locating the last node in the list, and then changing its `.next` field

from NULL to point to the new node, such as the `tail` variable in the following example of adding a "3" node to the end of the list {1, 2}...



This is just a special case of the general rule: to insert or delete a node inside a list, you need a pointer to the node just *before* that position, so you can change its `.next` field. Many list problems include the sub-problem of advancing a pointer to the node before the point of insertion or deletion. The one exception is if the node is the first in the list — in that case the head pointer itself must be changed. The following examples show the various ways code can handle the single head case and all the interior cases...

5) Build — Special Case + Tail Pointer

Consider the problem of building up the list {1, 2, 3, 4, 5} by appending the nodes to the tail end. The difficulty is that the very first node must be added at the head pointer, but all the other nodes are inserted after the last node using a tail pointer. The simplest way to deal with both cases is to just have two separate cases in the code. Special case code first adds the head node {1}. Then there is a separate loop that uses a tail pointer to add all the other nodes. The tail pointer is kept pointing at the last node, and each new node is added at `tail->next`. The only "problem" with this solution is that writing separate special case code for the first node is a little unsatisfying. Nonetheless, this approach is a solid one for production code — it is simple and runs fast.

```
struct node* BuildWithSpecialCase() {
   struct node* head = NULL;
   struct node* tail;
   int i;

   // Deal with the head node here, and set the tail pointer
   Push(&head, 1);
   tail = head;

   // Do all the other nodes using 'tail'
   for (i=2; i<6; i++) {
      Push(&(tail->next), i); // add node at tail->next
      tail = tail->next;      // advance tail to point to last node
   }

   return(head);  // head == {1, 2, 3, 4, 5};
}
```

6) Build — Dummy Node

Another solution is to use a temporary dummy node at the head of the list during the computation. The trick is that with the dummy, every node appear to be added after the `.next` field of a node. That way the code for the first node is the same as for the other nodes. The tail pointer plays the same role as in the previous example. The difference is that it now also handles the first node.

```
struct node* BuildWithDummyNode() {
   struct node dummy;   // Dummy node is temporarily the first node
   struct node* tail = &dummy; // Start the tail at the dummy.
                               // Build the list on dummy.next (aka tail->next)
   int i;

   dummy.next = NULL;

   for (i=1; i<6; i++) {
      Push(&(tail->next), i);
      tail = tail->next;
   }

   // The real result list is now in dummy.next
   // dummy.next == {1, 2, 3, 4, 5};
   return(dummy.next);
}
```

Some linked list implementations keep the dummy node as a permanent part of the list. For this "permanent dummy" strategy, the empty list is not represented by a NULL pointer. Instead, every list has a dummy node at its head. Algorithms skip over the dummy node for all operations. That way the heap allocated dummy node is always present to provide the above sort of convenience in the code.

Our dummy-in-the stack strategy is a little unusual, but it avoids making the dummy a permanent part of the list. Some of the solutions presented in this document will use the temporary dummy strategy. The code for the permanent dummy strategy is extremely similar, but is not shown.

7) Build — Local References

Finally, here is a tricky way to unifying all the node cases without using a dummy node. The trick is to use a local "reference pointer" which always points to the last *pointer* in the list instead of to the last node. All additions to the list are made by following the reference pointer. The reference pointer starts off pointing to the head pointer. Later, it points to the `.next` field *inside* the last node in the list. (A detailed explanation follows.)

```
struct node* BuildWithLocalRef() {
   struct node* head = NULL;
   struct node** lastPtrRef= &head; // Start out pointing to the head pointer
   int i;

   for (i=1; i<6; i++) {
      Push(lastPtrRef, i);    // Add node at the last pointer in the list
      lastPtrRef= &((*lastPtrRef)->next); // Advance to point to the
                                          // new last pointer
   }

   // head == {1, 2, 3, 4, 5};
   return(head);
}
```

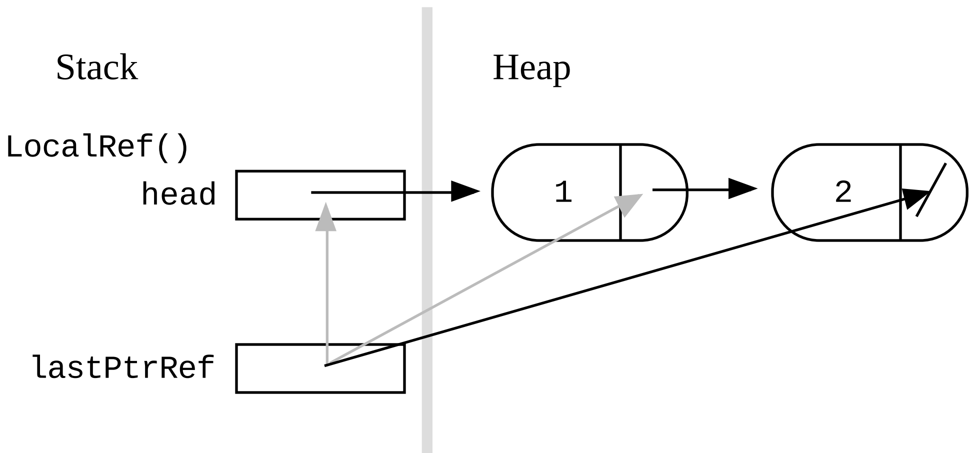
This technique is short, but the inside of the loop is scary. This technique is rarely used. (Actually, I'm the only person I've known to promote it. I think it has a sort of compact charm.) Here's how it works...

1) At the top of the loop, `lastPtrRef` points to the last pointer in the list. Initially it points to the head pointer itself. Later it points to the `.next` field inside the last node in the list.

2) `Push(lastPtrRef, i);` adds a new node at the last pointer. The new node becaomes the last node in the list.

3) `lastPtrRef= &((*lastPtrRef)->next);` Advance the `lastPtrRef` to now point to the `.next` field inside the new last node — that `.next` field is now the last pointer in the list.

Here is a drawing showing the state of memory for the above code just before the third node is added. The previous values of `lastPtrRef` are shown in gray...



This technique is never required to solve a linked list problem, but it will be one of the alternative solutions presented for some of the advanced problems.

Both the temporary-dummy strategy and the reference-pointer strategy are a little unusual. They are good ways to make sure that you really understand pointers, since they use pointers in unusual ways.

Section 4 — Examples

This section presents some complete list code to demonstrate all of the techniques above. For many more sample problems with solutions, see CS Education Library #105, -- Linked List Problems (http://cslibrary.stanford.edu/105/).

AppendNode() Example

Consider a AppendNode() function which is like Push(), except it adds the new node at the tail end of the list instead of the head. If the list is empty, it uses the reference pointer to change the head pointer. Otherwise it uses a loop to locate the last node in the list. This version does not use Push(). It builds the new node directly.

```
struct node* AppendNode(struct node** headRef, int num) {
   struct node* current = *headRef;
   struct node* newNode;

   newNode = malloc(sizeof(struct node));
   newNode->data = num;
   newNode->next = NULL;

   // special case for length 0
   if (current == NULL) {
      *headRef = newNode;
   }
   else {
      // Locate the last node
      while (current->next != NULL) {
         current = current->next;
      }

      current->next = newNode;
   }
}
```

AppendNode() With Push()

This version is very similar, but relies on Push() to build the new node. Understanding this version requires a real understanding of reference pointers.

```
struct node* AppendNode(struct node** headRef, int num) {
   struct node* current = *headRef;

   // special case for the empty list
   if (current == NULL) {
      Push(headRef, num);
   } else {
      // Locate the last node
      while (current->next != NULL) {
         current = current->next;
      }

      // Build the node after the last node
      Push(&(current->next), num);
   }
}
```

CopyList() Example

Consider a CopyList() function that takes a list and returns a complete copy of that list. One pointer can iterate over the original list in the usual way. Two other pointers can keep track of the new list: one head pointer, and one tail pointer which always points to the last node in the new list. The first node is done as a special case, and then the tail pointer is used in the standard way for the others...

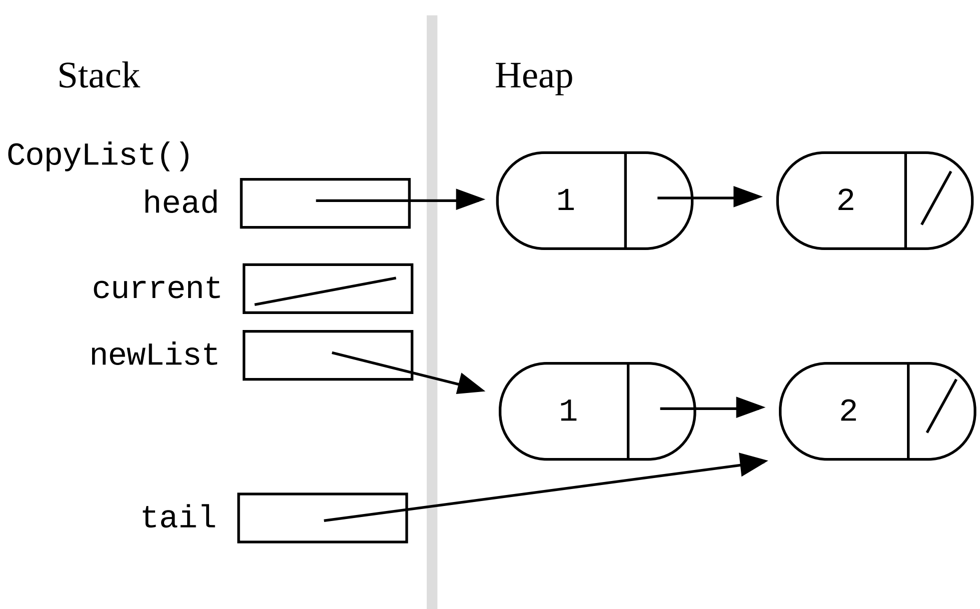
```
struct node* CopyList(struct node* head) {
   struct node* current = head;     // used to iterate over the original list
   struct node* newList = NULL;     // head of the new list
   struct node* tail = NULL;  // kept pointing to the last node in the new list

   while (current != NULL) {
      if (newList == NULL) {  // special case for the first new node
         newList = malloc(sizeof(struct node));
         newList->data = current->data;
         newList->next = NULL;
         tail = newList;
      }
      else {
         tail->next = malloc(sizeof(struct node));
         tail = tail->next;
         tail->data = current->data;
         tail->next = NULL;
      }
      current = current->next;
   }

   return(newList);
}
```

CopyList() Memory Drawing

Here is the state of memory when CopyList() finishes copying the list {1, 2}...



CopyList() With Push() Exercise

The above implementation is a little unsatisfying because the 3-step-link-in is repeated — once for the first node and once for all the other nodes. Write a CopyList2() which uses Push() to take care of allocating and inserting the new nodes, and so avoids repeating that code.

CopyList() With Push() Answer

```
// Variant of CopyList() that uses Push()
struct node* CopyList2(struct node* head) {
   struct node* current = head;     // used to iterate over the original list
   struct node* newList = NULL;     // head of the new list
   struct node* tail = NULL;  // kept pointing to the last node in the new list

   while (current != NULL) {
      if (newList == NULL) {  // special case for the first new node
         Push(&newList, current->data);
         tail = newList;
      }
      else {
         Push(&(tail->next), current->data);    // add each node at the tail
         tail = tail->next;   // advance the tail to the new last node
      }
      current = current->next;
   }

   return(newList);
}
```

CopyList() With Dummy Node

Anther strategy for CopyList() uses a temporary dummy node to take care of the first node case. The dummy node is temporarily the first node in the list, and the tail pointer starts off pointing to it. All nodes are added off the tail pointer.

```
// Dummy node variant
struct node* CopyList(struct node* head) {
   struct node* current = head;     // used to iterate over the original list
   struct node* tail;   // kept pointing to the last node in the new list
   struct node dummy;   // build the new list off this dummy node

   dummy.next = NULL;
   tail = &dummy;       // start the tail pointing at the dummy

   while (current != NULL) {
      Push(&(tail->next), current->data); // add each node at the tail
      tail = tail->next;      // advance the tail to the new last node
      }
      current = current->next;
   }

   return(dummy.next);
}
```

CopyList() With Local References

The final, and most unusual version uses the "local references" strategy instead of a tail pointer. The strategy is to keep a `lastPtr` that points to the last pointer in the list. All node additions are done at the `lastPtr`, and it always points to the last pointer in the

list. When the list is empty, it points to the head pointer itself. Later it points to the `.next` pointer inside the last node in the list.

```
// Local reference variant
struct node* CopyList(struct node* head) {
   struct node* current = head;     // used to iterate over the original list
   struct node* newList = NULL;
   struct node** lastPtr;

   lastPtr = &newList;        // start off pointing to the head itself

   while (current != NULL) {
      Push(lastPtr, current->data);       // add each node at the lastPtr
      lastPtr = &((*lastPtr)->next);      // advance lastPtr
      current = current->next;
   }

   return(newList);
}
```

CopyList() Recursive

Finally, for completeness, here is the recursive version of CopyList(). It has the pleasing shortness that recursive code often has. However, it is probably not good for production code since it uses stack space proportional to the length of its list.

```
// Recursive variant
struct node* CopyList(struct node* head) {
   if (head == NULL) return NULL;
   else {
      struct node* newList = malloc(sizeof(struct node));   // make the one node
      newList->data = current->data;

      newList->next = CopyList(current->next);  // recur for the rest

      return(newList);
   }
}
```

Appendix — Other Implementations

There are a many variations on the basic linked list which have individual advantages over the basic linked list. It is probably best to have a firm grasp of the basic linked list and its code before worrying about the variations too much.

- *Dummy Header* Forbid the case where the head pointer is NULL. Instead, choose as a representation of the empty list a single "dummy" node whose .data field is unused. The advantage of this technique is that the pointer-to-pointer (reference parameter) case does not come up for operations such as Push(). Also, some of the iterations are now a little simpler since they can always assume the existence of the dummy header node. The disadvantage is that allocating an "empty" list now requires

allocating (and wasting) memory. Some of the algorithms have an ugliness to them since they have to realize that the dummy node "doesn't count." (editorial) Mainly the dummy header is for programmers to avoid the ugly reference parameter issues in functions such as Push(). Languages which don't allow reference parameters, such as Java, may require the dummy header as a workaround. (See the "temporary dummy" variant below.)

- *Circular* Instead of setting the .next field of the last node to NULL, set it to point back around to the first node. Instead of needing a fixed head end, any pointer into the list will do.

- *Tail Pointer* The list is not represented by a single head pointer. Instead the list is represented by a head pointer which points to the first node and a tail pointer which points to the last node. The tail pointer allows operations at the end of the list such as adding an end element or appending two lists to work efficiently.

- *Head struct* A variant I like better than the dummy header is to have a special "header" struct (a different type from the node type) which contains a head pointer, a tail pointer, and possibly a length to make many operations more efficient. Many of the reference parameter problems go away since most functions can deal with pointers to the head struct (whether it is heap allocated or not). This is probably the best approach to use in a language without reference parameters, such as Java.

- *Doubly-Linked* Instead of just a single .next field, each node incudes both .next and .previous pointers. Insertion and deletion now require more operations. but other operations are simplified. Given a pointer to a node, insertion and deletion can be performed directly whereas in the singly linked case, the iteration typically needs to locate the point just *before* the point of change in the list so the `.next` pointers can be followed downstream.

- *Chunk List* Instead of storing a single client element in each node, store a little constant size array of client elements in each node. Tuning the number of elements per node can provide different performance characteristics: many elements per node has performance more like an array, few elements per node has performance more like a linked list. The Chunk List is a good way to build a linked list with good performance.

- *Dynamic Array* Instead of using a linked list, elements may be stored in an array block allocated in the heap. It is possible to grow and shrink the size of the block as needed with calls to the system function realloc(). Managing a heap block in this way is a fairly complex, but can have excellent efficiency for storage and iteration., especially because modern memory systems are tuned for the access of contiguous areas of memory. In contrast, linked list can actually be a little inefficient, since they tend to iterate through memory areas that are not adjacent.